Levenshtein Distance

Name: Xizhen Huang

Matrikel-Nr.: 4889117

Study Program: Distributed Systems Engineering

Email: Xizhen.Huang@mailbox.tu-dresden.de

Table of Contents

1. Description

1.1 Description of the problem

The Levenshtein Distance is a string metric for measuring the difference between two sequences. Informally, the Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

The Levenshtein distance between two strings *a, b* (of length $|a|$ and $|b|$ respectively) is given by $\mathrm{lev}_{a,b}(|a|, |b|)$ where

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

where $1_{(ai\ !=\ bj)}$ is the indicator function equal to 0 when $a_i = b_j$ and equal to 1 otherwise, and $\mathrm{lev}_{a,b}(i, j)$ is the distance between the first i characters of a and the first j characters of b. i and j are 1-based indices.

In this task, unmodified algorithm needs to be optimized and covered into parallel code, working on 56 cores machine.

1.2 Description of unmodified algorithm

The unmodified algorithm uses recursion to solve problem, straightforward but inefficient, because it recomputes the Levenshtein Distance of the same substrings many times.

Following is the key part of this algorithm.

```
if (f1 == l1) return dist (f2, l2) ;    // if string1 is null, return length of string2
if (f2 == l2) return dist (f1, l1) ;    // if string2 is null, return length of string1
cost = (*pred (l1) == *pred (l2)) ? D (0) : D (1) ;     // if ai = bj, cost=1, otherwise 0
return min                                              // recursion part
        ( lev_dist (f1, pred (l1), f2,       l2 ) + D (1)
        , lev_dist (f1,       l1,  f2, pred (l2)) + D (1)
        , lev_dist (f1, pred (l1), f2, pred (l2)) + cost ) ;
```

1.3 Complexity analysis of the existing algorithm

The time complexity is exponentially increasing. The complexity of the problem is **O(3^n)**, and many of the same sub-problems have been solved many times. Using dynamic programming can reduce the complexity to **O(m*n)**, m and n are lengths of two strings.

2. Optimization Strategy

Because recursion algorithm is inefficient, so dynamic programming is used to improve performance. if reserve a matrix to hold the Levenshtein Distances between all prefixes of the first string and all prefixes of the second, then values in the matrix could be computed in a dynamic programming fashion, and thus find the distance between the two full strings as the last value computed.

$$Matrix[i][j] = min(Matrix[i-1][j] +1, Matrix[i][j-1] +1, Matrix[i-1][j-1] +cost)$$

$$String1[i] == String2[j], Cost = 0$$

$$String1[i] == String2[j], Cost = 1$$

Example of the result matrix, the bottom-right element of the array contains the answer. And the two parallelization strategies described in section 3 are based on this strategy.

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

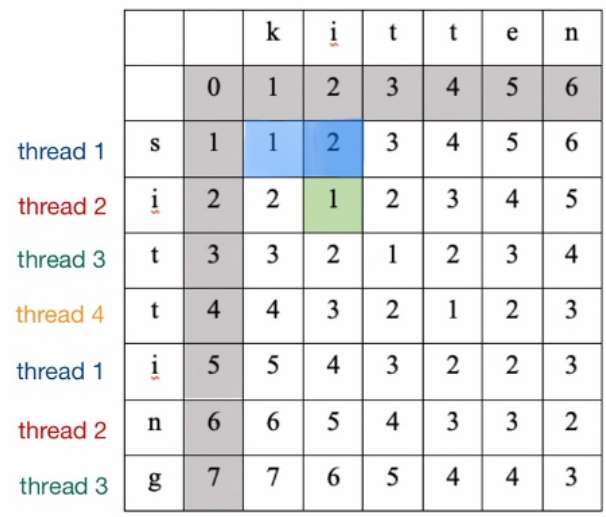
Table 1. resulting matrix of two strings

3. Parallelization strategies

3.1 Lock-Free & Busy-waiting

The lock-free mechanism defines a critical section which could be visited and modified by all threads. In this strategy, a global two-dimensional array *matrix* is defined to store results. All threads access it without locking.

In this strategy, each row of the matrix is a task, a thread may have one or more tasks depends on the number of threads. Following figure shows the situation which 4 threads are created. The matrix is divided into 7 tasks, thread 1 is responsible for computing task 1 and task 5.



		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

Figure 1. lock-free & busy-waiting working pattern

According to Table 1 (in section 2), it is known that if want to compute value of *matrix[i][j]*, the correct value of *matrix[i-1][j], matrix[i][j-1]* and *matrix[i-1][j-1]* must be known in advanced. So, if thread 2 want to compute the value of matrix [2][2] (the green grid), it has to wait until calculations of matrix [1][2] and matrix [1][1] have been completed by thread 1 (the blue grids).

3.2 Lock-Free & Synchronization

The lock-free mechanism defines a critical section which could be visited and modified by all threads. In this strategy, a global two-dimensional array *matrix* is defined to store results. All threads access it without locking.

In this strategy, a grid is a task instead of a row in the matrix. Depend on the number of threads and the number of grids in current anti-diagonal, a thread may be assigned zero, one or more tasks, and the calculation process is repeated ***m+n+1*** times, because there are ***m+n+1*** anti-diagonals (m is the length of string1, n is the length of string2). Three variables – rounds, the number of grids in this anti-diagonal, sync barrier – are keys of this parallelization algorithm. Barrier guarantees threads synchronization.

Following figure shows the situation which 4 threads are created. In round 5, 4 tasks need to compute, and 6 tasks need to compute in round 7.

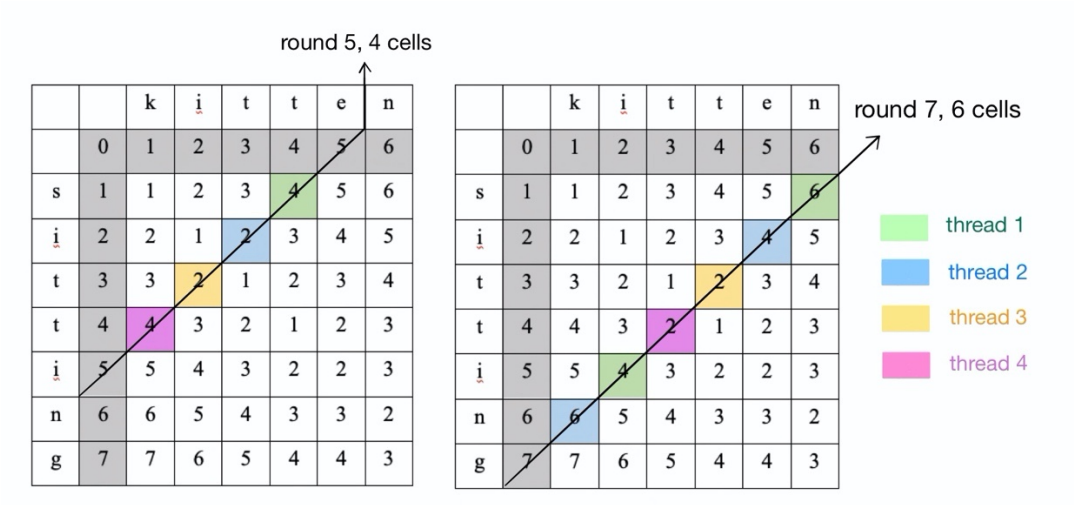


Figure 2. Lock free & Synchronization working pattern

4. Evaluation of parallelization

4.1 Evaluation of Lock-Free & Busy-waiting

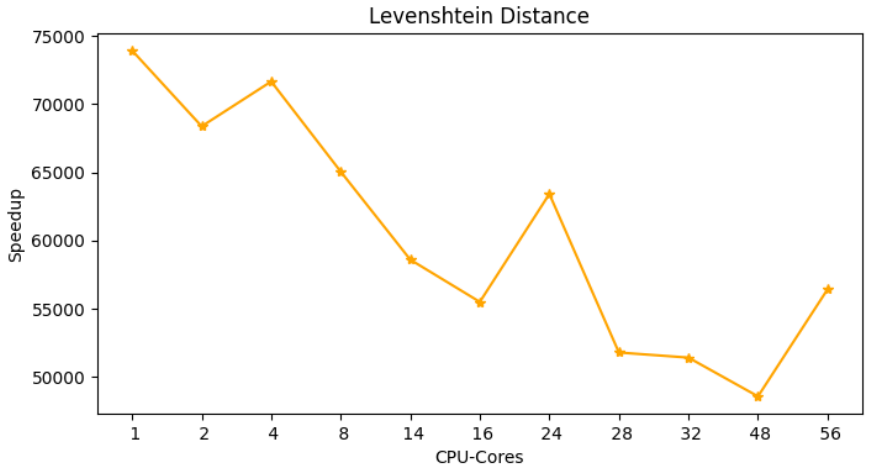


Figure 3. Speedup of optimization strategy 1

Multithreads is used in this algorithm to achieve parallel execution. It can be seen speedup decreasing with more threads in this graph, because threads depend on each other's work. In this strategy, if the thread has not obtained the required data, it will enter a busy-waiting state. Using Go pprof to analyze code, it shows that waiting for required data takes the longest time.

```
Total:      1.44s      1.56s (flat, cum) 73.93%
    17          .          .
    18          .          .                        r = id + workerNums*time
    19          .          .                        for c = 1; c < col+1; c++ {
    20          .          .
    21          .          .                                // wait precondition
    22      1.44s      1.45s                                for matrix[r-1][c] == -1 {
    23          .          .                                }
    24          .          .
    25          .          .                                if s[r-1] == t[c-1] {
    26          .          .                                        matrix[r][c] = matrix[r-1][c-1]
    27          .          .                                } else {
    28          .          .                                        matrix[r][c] = minimum(matrix[r-1][c]+1, matrix[r][c-1]+1, matrix[r-1][c-1]+1)
    29          .          .                                }
    30          .          .                        }
    31          .          .                }
    32          .      110ms                wg.Done()
    33          .          .            }
```

Figure 4. Runtime analyzation strategy1

Accroding to Amdahl's Law,

$$S_{latency}(s) = \frac{1}{1-p+\frac{p}{s}}$$

where is the theoretical speedup of the execution of the whole task, is the speedup of the part of the task that benefits from improved system resources and is the proportion of execution time that the part benefiting from improved resources originally occupied. It can be calculated from the evaluation of the strategy that in the promising strategy, the parallelization part accounts for 103% of the whole execution time in 28 cores, 101% in 56 cores. p value exceeds 100%, because the speedup is compared with the unmodified code execution duration, and the optimization algorithm provides most part of the acceleration, instead of parallelization.

In this code, parallelization part is computing the value of every grids in per row. Serialization part including threads initialization, task assignment. The more threads, the more overhead in this part.

4.2 Evaluation of Lock free & Synchronization

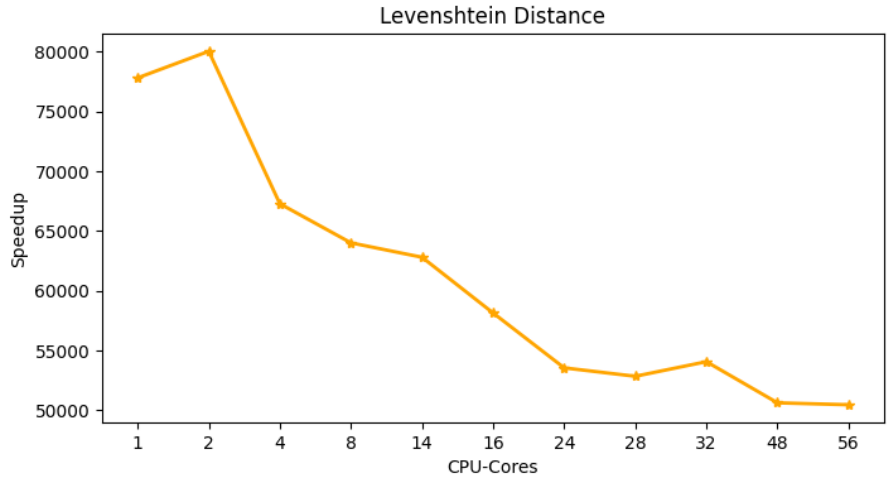


Figure 5. Speedup of optimization strategy 2

Multithreads is used in this algorithm to achieve parallel execution. It can be seen speedup decreasing with more threads in this graph, because in this strategy, before getting into next stage, threads need to be synchronized. Using Go pprof to analyze code, it shows that semaphore_wait_group takes the longest time.

/usr/local/Cellar/go/1.10.8/libexec/go/src/runtime/sys_darwin_amd64.s

```
Total:       1.59s      1.59s (flat, cum) 52.13%
   535            .          .           // func mach_semaphore_wait(sema uint32) int32
   536            .          .           TEXT runtime·mach_semaphore_wait(SB),NOSPLIT,$0
   537            .          .                 MOVL    sema+0(FP), DI
   538            .          .                 MOVL    $(0x1000000+36), AX     // semaphore_wait_trap
   539            .          .                 SYSCALL
   540        1.59s      1.59s                 MOVL    AX, ret+8(FP)
   541            .          .                 RET
   542            .          .
   543            .          .           // func mach_semaphore_timedwait(sema, sec, nsec uint32) int32
   544            .          .           TEXT runtime·mach_semaphore_timedwait(SB),NOSPLIT,$0
   545            .          .                 MOVL    sema+0(FP), DI
```

Figure 6. Runtime analysis of strategy2

Accroding to Amdahl's Law,

$$S_{latency}(s) = \frac{1}{1-p+\frac{p}{s}}$$

The parallelization part accounts for 103% of the whole execution time in 28 cores, 102% in 56 cores. p value exceeds 100%, because the speedup is compared with the unmodified code execution duration, and the optimization algorithm provides most part of the acceleration, instead of parallelization.

In this code, parallelization part is computing the value of every grids in per anti-diagonal. Serialization part including threads initialization, task assignment. The more threads, the more overhead in this part.

5. Comparison of the parallelization strategies

Both parallelization strategies are scalable, but scalability is not good. In both strategies, as the amount of data to be calculated increases, the tasks assigned to each thread will also increase, but workload for each thread is almost the same.

The dependence between threads can’t be avoided, so both algorithms have the same problem – waiting time will increase with more threads.

Appendix

Measurement	Git Hash	Evaluation URL
Baseline	5f02e6f4	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-25T09:16:26+02:00.log
Scalability Evaluation (busy waiting)	7394271f	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-23T21:35:56+02:00.log
Scalability Evaluation (synchronization)	39eb2509	https://cds-lab.pages.se-gitlab.inf.tu-dresden.de/cds-s-2020/cds-website/logs/a0967e72cdb1108127661224e04c5f6bcac4d456d52090209b451d36429f223b/2020-06-27T19:21:04+02:00.log